Suppose that you and I work for a small business, and that we're designing a replacement for an unwieldy time tracking system that has been in use for almost a decade.

I was the main developer for the original application, which replaced a tedious paper process that everyone hated. The software has mostly served its purpose over the years; but a decade of constant use has made its weak spots blindingly obvious. In particular, the application's core data model has many limitations that we'd like to avoid repeating in any new system we build.

Because you weren't involved in developing the original application, you don't have a ton of context on how it evolved over time. I see this as a good thing, because it means you'll be able to look at the problem domain with a fresh pair of eyes, and help me see beyond my own blind spots.

For this reason, I've asked you to take the lead on designing the core data model for our replacement system. You've spent the last few days reading the old codebase and playing around with ideas, and today you'll present your plan for building a better system. Let's get started!

Decouple conceptual modeling from physical modeling

You kick things off with a rough sketch of a typical employee's daily interactions with the time tracking system:

- 8:30 AM: Clock in at the start of the work day.
- 1:30 PM: Clock out when leaving for lunch.
- 2:30 PM: Clock in after returning from lunch.
- 5:15 PM: Clock out at the end of the work day.
In the current application, this sequence of events would create a pair of `WorkSession` records, which are modeled as intervals in the database. One would run from 8:30 AM to 1:30 PM, and another from 2:30 PM to 5:15 PM.

You point out that this design makes sense as a conceptual model, but that it unnecessarily complicates raw data manipulation. I ask you to give an example, and you're more than happy to oblige:

**YOU:** Suppose an employee forgets to clock in at 8:30 AM, but remembers all the rest of there punches for the day. What intervals will be created then?

**GTB:** Hmm... it'll misinterpret the 1:30 PM punch as an IN punch, because that would be the first punch entered for the employee on that day. From there, it'll create an interval from 1:30 PM - 2:30 PM, and then another that starts at 5:15 PM but leaves its finish time undefined.

**YOU:** Exactly! Until this error gets corrected, the data itself will be out of sync with reality, in a pretty confusing way. To make things worse, editing the data to get things back into a consistent state would involve touching four separate fields spread across two records, which is a confusing process.

You quickly mock up an example to illustrate your point:

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
<th>WORKED</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 AM</td>
<td>1:30 PM</td>
<td>5.00 HRS</td>
</tr>
<tr>
<td>1:30 PM</td>
<td>2:30 PM</td>
<td>1.00 HRS</td>
</tr>
<tr>
<td>2:30 PM</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>5:15 PM</td>
<td>NULL</td>
<td>2.75 HRS</td>
</tr>
</tbody>
</table>

You also mention that you've taken a look at how often missed punches happen in the system, and that the data shows that this is a daily headache for the management staff.

I spend a moment thinking about the problem, and then I realize it is a relatively recent one. Up until a couple years ago, employees only punched a clock twice a day. In that original workflow, the employee would have punched in at 8:30 AM and out at 5:15 PM, then a predefined amount of lunch time would automatically be deducted from their hours. With only a single IN punch and OUT punch per day, times could be edited as needed without any complications.

Seven years after the system was created, company policy changed and it became mandatory to record punch times for all breaks. The system needed to be updated to support this new rule -- but it was done with a shoestring budget, and by that time the codebase had degraded in the way that legacy projects left untouched for years...
often do. Under those constraints, the idea of improving the workflow to fit the new requirements was out of the question.

You've identified this design flaw as a problem that is worth fixing in the new system, and have a specific solution in mind:

**YOU:** My goal is to make it so that you never need to modify a good punch in order to correct an error. If you need to add a missing punch, it should be possible to key in that one missed time, and leave everything else untouched.

In order to make this possible, we'd stop modeling work sessions as intervals in the database, and instead record punches as individual events. From there, we'd convert raw punch data into intervals at the application level whenever we need to display a report or run a computation.

**GTB:** This won't change the fact that if a punch is missed, the timesheet will still show incorrect IN/OUT pairs until it gets corrected.

**YOU:** That's true, although in the current system it isn't just the reports that can break -- the data itself gets corrupted. So you end up editing a bunch of fields just because the system put the punch data in the wrong place.

With the new model, basic facts of the system stay accurate even if the reporting still gets out of sync with reality in certain edge cases. Whenever an employee records a punch, that actually happened and so the system can treat it as a fact. Work sessions generated from those punches are a more muddy concept, and this approach cleanly separates the two.

**GTB:** Got it! Then yeah, this makes sense. I still would like to see an example of how the new model will work, though.

You draft up another quick figure to demonstrate that in the new model, a missing punch can be directly added to the punch list with no other modifications needed:

```
<table>
<thead>
<tr>
<th>Punches</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 AM</td>
</tr>
<tr>
<td>1:30 PM</td>
</tr>
<tr>
<td>2:30 PM</td>
</tr>
<tr>
<td>5:15 PM</td>
</tr>
</tbody>
</table>
```

From here, the data would be converted into `WorkSession` objects at the application level by creating a new interval for each consecutive pair of punches. But because these intervals would be dynamically generated at runtime, no special consideration needs to be given to them when the raw punch data is updated.
In a system with messy data sources, it’s often better to preserve some degree of flexibility by not imposing too much structure at the physical data modeling level. Your examples illustrate that point nicely, so we’re now ready to move on to your next planned improvement.

**Prefer an explicit model for tracking data changes**

You’ve asked me to explain how the existing time tracking system implements its audit logs. Only so much can be guessed at when staring at code that was hastily written a decade ago, so I’m happy to fill you in on the back story.

It was well understood from day one that the software needed a comprehensive audit trail for any changes to time records. The data in this application corresponds directly to what employees get paid, so the potential for fraudulent management activity couldn’t be ignored.

On top of this, we also knew that on occasion we’d need to go back and examine original punch data from time to time, to catch discrepancies in employee records. For example, if someone consistently misses punches or requests that later punch times be adjusted to reflect an earlier start time, it could be a sign of a problem.

In the best case scenario, this might mean figuring out what the source of distraction is that is causing the employee to begin working before clocking in. But in the worst case scenario, this pattern of behavior could be a sign of someone intentionally falsifying their time records. In either case, an audit log is important because it helps with detecting the problem and can also serve as evidence later as to why the behavior is being questioned in the first place.

For the most part, the auditing requirements were understood to be protections against truly exceptional circumstances. In the entire history of the application, we’ve only needed to dig into these logs a handful of times. With that in mind, we took a very minimal approach towards implementing these features.

Instead of designing a custom-built auditing system, we instead made use of a third party library that worked at the database layer to track changes to the tables we wanted to maintain historical records for. The basic metaphor behind the whole thing was that of a backup mechanism:

- Create a read-only copy of the record before it is modified.
- Update the record with whatever changes you need to make.
- Update the admin_td field to indicate who approved the change.
- Increment the record’s version number.

The copied records are stored in their own versions table, but have all the necessary information to review changes between versions, or to revert to an old version if necessary. The main caveat is that because this is all done at the database level, the con-
cept of a revision is tied to a record insert or update, rather than a meaningful business transaction.

To demonstrate how the versioning mechanism would work for the “add a missed punch at 8:30 AM” scenario we’ve been using throughout our discussion, I cook up the following example:

<table>
<thead>
<tr>
<th>Punches</th>
<th>Work Session Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 AM</td>
<td></td>
</tr>
<tr>
<td>1:30 PM</td>
<td></td>
</tr>
<tr>
<td>2:30 PM</td>
<td></td>
</tr>
<tr>
<td>5:15 PM</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>SESSION_ID</th>
<th>IN</th>
<th>OUT</th>
<th>ADMIN_ID</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1:30 PM</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>1:30 PM</td>
<td>2:30 PM</td>
<td>NULL</td>
<td>2</td>
</tr>
<tr>
<td>1001</td>
<td>8:30 AM</td>
<td>1:30 PM</td>
<td>1234</td>
<td>3</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1002</td>
<td>5:15 PM</td>
<td>NULL</td>
<td>NULL</td>
<td>1</td>
</tr>
<tr>
<td>1002</td>
<td>2:30 PM</td>
<td>5:15 PM</td>
<td>1234</td>
<td>2</td>
</tr>
</tbody>
</table>

I try to explain how this works, knowing that it’s a confusing process that you’ll have no trouble poking holes in right away.

**GTB:** In order to add the missed punch, a new version gets created for each of the two work sessions. And you can see from the data that these changes were made by a manager, because they include an `ADMIN_ID`.

**YOU:** But how do you show that these two changed records are actually part of a single change request?

**GTB:** You can’t. Not directly from the data, anyway. You’d have to pull the full history of work sessions for an employee on a particular day and just infer what happened from the changes that were made.

**YOU:** So... you mean like noticing that the 2:30 PM OUT time in version 2 of session 1001 ends up becoming the IN time in version 2 of session 1002?

**GTB:** Uh... yeah. This is super confusing, and the few times I’ve needed to run reports against it I’ve scratched my head for a while until I sorted out what happened, and then I’d send a clean report to the management team. The data itself is a giant mess, though. I’m thankful that the need to work with it has been so rare that I haven’t had to think about this much.

**YOU:** As bad as this is, I imagine it easily gets even worse. What happens if there’s a typo when the manager enters a revised time, and then they go back and correct it later? Does that create a new version too, even if they fix it immediately after submitting the change?

My awkward silence makes it clear that I’m painfully aware of my past mistakes, and that I’m ready to hear your new path forward.
You start describing a design where the audit log isn’t a bolt-on feature implemented at the database level, but instead is explicitly modeled as part of the business domain.

Knowing that some sample data will help me understand better, you then show me the following example:

<table>
<thead>
<tr>
<th>Punches</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 AM</td>
</tr>
<tr>
<td>9:17 AM</td>
</tr>
<tr>
<td>1:30 PM</td>
</tr>
<tr>
<td>2:30 PM</td>
</tr>
<tr>
<td>5:15 PM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timesheet Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKDAY</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>2016-03-17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Punch Adjustments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>add</td>
</tr>
</tbody>
</table>

From this rough sketch, I learn that a `TimesheetRevision` represents the high level information about a change: which workday it is for, a note explaining why the change is needed, a reference to the admin who approved the change, etc. I also can see that the `PunchAdjustment` model represents the individual punch that needs to be added to the timesheet.

You then show me a couple examples of how your new model would support a couple other change requests that we’ve frequently seen from employees.

An employee who forgets to punch in until after an early morning meeting:

<table>
<thead>
<tr>
<th>Punches</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 AM</td>
</tr>
<tr>
<td>9:17 AM</td>
</tr>
<tr>
<td>1:30 PM</td>
</tr>
<tr>
<td>2:30 PM</td>
</tr>
<tr>
<td>5:15 PM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timesheet Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORKDAY</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>2016-03-17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Punch Adjustments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>remove</td>
</tr>
</tbody>
</table>
An employee who forgets to record their lunchtime punches:

<table>
<thead>
<tr>
<th>WORKDAY</th>
<th>NOTE</th>
<th>ADMIN_ID</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-03-17</td>
<td>Forgot to record punches for lunch</td>
<td>1234</td>
<td>1001</td>
</tr>
</tbody>
</table>

These examples tell me a lot about what you're trying to do with this new design, but there are still a few open questions that need to be sorted out.

**GTB**: Overall, I agree that this approach makes the audit trail easier to understand. But what else does this do for us?

**YOU**: Honestly, I modeled things this way at first just to clean up the auditing system, but then I realized that it can enable a much better workflow for administrators.

**GTB**: How so? I can't really see that from what you've shown so far.

**YOU**: In the existing system, `WorkSession` records are edited directly, and as a side effect of this, a read-only copy of the record was made before any modifications were applied.

But when you edit multiple `WorkSession` records simultaneously as part of a single change, there's no easy way to tie them together. This limits (or at least complicates) the kinds of features we can implement that would make the punch editing process less error prone.

**GTB**: Can you be more specific? Keep in mind that I probably have serious tunnel vision here, because I've been thinking in terms of how the old system is implemented for over a decade now.

**YOU**: Sure! Wouldn't it be nice if you could review the pending changes to a timesheet before updating the official records?

If we use the `TimesheetRevision` model to generate a live preview, any mistakes could be corrected before the changes were committed and signed off on.

**GTB**: Hmm... yes! That would be useful. Now I think I get why you modeled things this way: you're planning to use the `TimesheetRevision` and `PunchAdjustment` models to drive changes to the `Punch` records, rather than the other way around.

**YOU**: Exactly. I'm attempting a rough approximation of the event sourcing pattern here. By representing the changes we'd like to make to the timesheet as a sequence of `PunchAdjustment` events, we can defer updates to the raw `Punch` data until later.
GTB: This sounds promising, but how will we deal with conflicts? I imagine if the data ever ends up in an inconsistent state, it'll be hard to figure out why these events don't end up generating the results we'd expect them to.

YOU: For starters, keep in mind that the Punch model in this new design is fundamentally more simple than the WorkSession model in the existing system. A Punch will only ever be created in two ways: either directly via the timeclock that employees use to clock in and out, or via a PunchAdjustment event that is approved by a manager.

Once a Punch is created, its timestamp is never modified. The only state transformation that can happen to a Punch is for it to be flagged as removed via a PunchAdjustment on an accepted TimesheetRevision. Once the removed flag is set, a Punch is completely hidden and won't be interacted with anymore.

From here, we can add the restriction that only a single pending TimesheetRevision can be open for an employee on a particular workday at any point in time. We'll need to see how that works in practice, but it seems like an acceptable constraint for now.

GTB: I'm not 100% sure that I understood all of that, but basically what you are saying is that because we capture all changes as well-defined events, and because we will not have multiple concurrent TimesheetRevision records open for a single employee/workday combination at the same time, the data should stay consistent by design?

YOU: More or less, yep! I don't expect you to take my word for it, but that's the underlying idea behind the event sourcing pattern, and as long as we implement it properly, this should work well for us.

I pause for a moment to think through how both of your recommended changes so far have a common theme: reducing incidental complexity as much as possible by reducing mutable state. There is probably a whole lot more than can be said about this, but you're already itching to move on to your next big idea.

**Understand how Conway’s Law influences data management practices**

“Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations” -- Melvin Conway

We start to talk about how timesheet change requests are handled by the business, and I come to realize that may be the biggest weak spot in the current workflow.

The process is completely adhoc -- each employee who needs a change communicates it to their manager by whatever method happens to be convenient for them: whether it's an in-person conversation, an email, a phone call, etc. The manager then reviews and aggregates these requests and sends them along to the payroll administrator, who makes the changes within the time management system.

The feedback cycle is variable, but tends to be on the slow side. Confirmation of a change can take days, and there tends to be a rush around the end of the pay period to get all the timesheets reconciled so that checks can be cut. If a request gets dropped
or some details about it get miscommunicated, it might take several passes through
the feedback loop to get corrected.

Employees have discovered a workaround for this broken process: if they submit
their requests directly to the payroll administrator through an internal messaging fea‐
ture within the time tracking system, they tend to get acted on quickly, with greater
accuracy. But in doing this, they cut their own managers out of the loop, and so this
isn’t ideal from an administrative perspective. Some employees file their requests in
duplicate in an attempt to both follow the rules and get a quick response, which
results in even more confusion.

The company is big enough where this messy process creates friction daily, but small
enough where fixing this problem hasn’t been a high priority. But it’s clear that an
improvement here would be welcome as long as it wasn’t too costly to implement.
Your theory is that this problem will be easy to solve in the new time tracking system,
because the new data model will open up doors that were previously closed to us.

YOU: I know this might be a hard sell, but I think the real way to fix this issue is to let
employees adjust their timesheets themselves.

GTB: I was afraid you might say that. I think it’s a great idea, but this is a really tough
topic to get into with management. I don’t even know where to begin the discussion on
this, because it is such a departure from how things are typically done around here.

YOU: Well, what do you think the major stumbling blocks are? What are they likely to
be most concerned about?

GTB: For starters, I think they’ll be worried about technical training issues. Part of the
reason why the payroll administrator keys in all the changes rather than having the
managers at each office take care of that is because early attempts to train management
staff on the timesheet editing process didn’t go well.

YOU: Not to be too harsh, but do you think the poor design of the original system had
anything to do with that? Editing four text fields just because you want to add a single
punch to a sheet seems awkward even for a programmer.

The old system also had no way to review your changes, no way to easily undo a
change, and no way to edit a whole day’s worth of times at once: you were brought to a
separate form for each time interval you edited -- a consequence of using an auto-
generated admin panel rather than building a custom interface.

GTB: So you’re saying it’s my user-hostile interface that is to blame? I’m not sure I
would have agreed a decade ago, but I’ve definitely changed my perspective since then.
But it’s also such a large timescale that expectations around human-centric design have
really shifted, even in business applications.

That said, there are still plenty of programs in use here that were built 20 years ago or
more, all of them which are even more awkward to use than the time tracking system.
All of this contributes to a generally uneasy relationship with software throughout the
company, and so even if we can convince them that we can build something learnable,
we’ll need to go beyond that to get their approval.
YOU: Well, what else do they really care about? If we know what matters to them, we can find a way to emphasize those points when suggesting a change to the workflow.

GTB: I know the management staff cares a great deal about accuracy, even they take a very messy approach towards obtaining it. The idea of overpaying or underpaying an employee because their timesheet wasn’t correct is especially bothersome to them, and rightly so.

Their theory is that if the payroll manager keys in all the changes, that leaves one person directly responsible for maintaining accurate records. By training that person well and making sure that they’re very careful, the potential for data entry errors is reduced. The payroll manager knows all the common mistakes people might make, and can follow up when a request looks problematic.

YOU: What do you think? Does that approach really work as well as they think it does?

GTB: I think given the limitations of the existing system, they have an effective process in place. The main problem I see is that this creates a huge amount of work for one person, and it’s unclear to me whether that’s really a cost effective way of doing things or not.

YOU: OK, I think I have a way to deal with these concerns. Before I get into that, what else do you think matters here?

GTB: Well, the other major recurring theme is a strong desire for effective oversight. Even minor discrepancies tend to be followed up on, as a proactive measure to limit fraud and abuse.

The tradeoff is that active monitoring tends to erode trust within the company, and also eats up time that managers might be able to spend on more important issues.

By giving some thought to cultural values that influence the way the company operates, we’re able to get a clearer picture of what our design constraints are.

We come to realize that for an improvement to the timesheet editing workflow to even be considered, the proposed alternative would need to be very easy to use, and it would need to make data entry errors easy to catch and correct. It would also need to preserve or even extend the effectiveness of management oversight built into the current process.

You claim that the idea you have in mind will meet all of these needs, and then some. I’m somewhat skeptical, but also excited to hear what you have in mind.

Remember that workflow design and data modeling go hand in hand

The two of us agree that allowing employees to modify their own timesheets could be a major improvement over the company’s current process, as long as we frame it in the right way.

To make things more concrete, you point out several specific benefits that would come along to switching to this new way of doing things:
• As long as employees are able to preview the changes they makes to their timesheet before submitting them, they will know exactly what to expect if and when their request has been approved. This will help prevent data entry errors due to miscommunications about what changes are needed.

• Assuming that all pending changes are clearly marked, timesheets and other reports can be immediately updated to reflect the requested changes rather than continuing to display incomplete or inaccurate information.

• Instead of relying on managers who are spread across half a dozen offices to aggregate requests and forward them to the payroll manager, all of the requests will be directly entered into the system by employees and the only step that will remain is to review and approve the changes. This will cut down on a huge amount of error-prone busy work for the management staff.

• If there is an open question about a particular change, all of the management staff as well as the employee who submitted a change request will be looking at the same information at the same time. If a request needs to be modified, then that too will be updated in realtime and visible to everyone who needs to see it.

• Because this new change request system would move official requests to modify timesheet data into the time tracking application itself, the paper trail would be far more complete and consistent than what is currently in place at the company.

• Notifications about whether a requested change have been accepted or declined by management could be automated, preventing the possibility of a decision being made without it being communicated.

• Warnings could be presented to the payroll manager whenever there are still pending timesheet editing requests at the end of a pay period.

All of these potential benefits hinge on having an implementation that works well enough to overcome the friction of change. You’ve done a few technical spikes around these ideas and sketched up some mockups, and so now it’s time to see what you’ve come up with.
At the heart of your solution is a presenter object that combines two key pieces of data: the punches that have already been committed for a particular workday, and the proposed adjustments to that list of punches.

This combined dataset will be used to present three important pieces of information: what the timesheet looked like before a requested change, what it will look like after the change is applied, and a summary view of exactly what the changes are.

You point out that the **AFTER** view of data isn’t just meant to be used to preview the final state of a request, but can also be updated live as individual PunchAdjustments are added to a TimesheetRevision. This makes it possible to mimic direct punch editing in the user interface:
When the employee is ready to submit their request, they are shown the before, after, and diff views side by side. After reviewing their changes to make sure they are accurate, they fill in a notes field explaining why the change is needed.

Once the request is submitted, it appears alongside all other open requests in the management panel, looking something like what is shown below:

If the request is approved, then a new Punch for 8:30 AM will be created, and the 9:17 AM Punch will be flagged as removed. If it is denied, then the TimesheetRevision will be closed without any Punch data being modified. In either case, the timesheet will return to a state in which there are no pending changes.

The critical feature of this workflow is that the official time records used for computing employee paychecks are only ever modified when the payroll manager approves a change. This achieves the same centralized control and oversight that is baked into the current process, but streamlines the communication around a change in a way that should greatly reduce data entry errors.
As I look back on your suggested improvements to the core data model, it becomes immediately apparent to me that each of them got us one step closer to building a better workflow. The striking thing about where we ended up is that it didn't take massive, earth-shattering changes to get here -- just relatively minor changes to the way that we store and interact with the data in the system.

There is no guarantee that our plans for this new workflow will be accepted by the business, the practical constraints of politics and budget always need to be considered. However, at least the two of us are convinced that we've come up with something truly better than what is already in use, and we're confident that at least some of these ideas will survive in the new system and make things better for all of its users.